Amortized Analysis

- Consider a data structure that supports various high-level operations
  - For example, a dictionary data structure supports INSERT, DELETE, FIND
- Often we focus on analyzing the worst-case cost of each of the individual operations
  - To obtain an upper bound the worst-case cost of a sequence of operations, we can the sum the worst-case cost of the individual operations in the sequence
  - Often it is impossible for every individual operation in a sequence of operations to have worst-case complexity
  - In such cases, we can use amortized analysis to try to establish a stronger upper bound

Methods of Amortized Analysis

- Assume that our data structure is initially in state D_0
- Operation i transforms the data structure from state D_{i-1} to state D_i, $1 \leq i \leq n$
- Let c_i denote the actual cost of operation i, $1 \leq i \leq n$
- We wish to upper bound $\sum_{1 \leq i \leq n} c_i$
- The text discusses three methods of amortized analysis
  - The aggregate method
  - The accounting method
  - The potential function method

Amortized Cost of an Operation

- We wish to assign an amortized cost $\hat{c}_i$ to each operation such that the following conditions hold for any sequence of n operations
- We have $\sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} \hat{c}_i$
- It is easy to evaluate $\sum_{1 \leq i \leq n} \hat{c}_i$
- Ideally, the latter sum is $\Theta(\sum_{1 \leq i \leq n} c_i)$

Aggregate Method

- Exploit the specific characteristics of the data structure under consideration to get an upper bound on the total actual cost
- Typically, our goal is to argue that for each "expensive" operation in a given sequence, we need to have many "cheap" operations
    - We can assign a low amortized cost to each expensive operation by assigning a slightly higher amortized cost to each cheap operation
    - This "spreads out" the high actual cost of an expensive operation over many cheap operations
    - Sometimes we end up with the same amortized cost for every operation, in which case it is easy to sum the amortized costs

Accounting Method

- In the accounting method, we augment the state of our data structure D (for the purposes of analysis only) by associating "tokens" with various elements of D
- For example, if D is a binary tree, we might maintain a pile of tokens at each node of the tree
    - The number of tokens in each pile must be nonnegative
- For a "cheap" operation, we assign an amortized cost higher than the actual cost, and we introduce a number of tokens that is at most the amortized cost minus the actual cost
- For an "expensive" operation, we assign an amortized cost lower than the actual cost, and we make up the difference by consuming a number of tokens that is at most the actual cost minus the amortized cost
- If there are no tokens in D_0, then $\sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} \hat{c}_i$

Potential Function Method

- In the potential function method, we augment the state of our data structure D with a single number $\Phi(D)$, called the potential
  - We require $\Phi(D)$ to be nonnegative
  - We define $\hat{c}_i$ as $c_i + \Phi(D_i) - \Phi(D_{i-1})$
  - Thus $\sum_{1 \leq i \leq n} \hat{c}_i$ is equal to $(\sum_{1 \leq i \leq n} c_i) + \Phi(D_n) - \Phi(D_0)$
- We typically define the potential function so that $\Phi(D_0) = 0$, and hence $\sum_{1 \leq i \leq n} c_i \leq \sum_{1 \leq i \leq n} \hat{c}_i$
- An accounting method analysis can be translated to the potential function method by defining $\Phi(D)$ as the total number of tokens in D

Example: Stack with MultiPop

- We have a stack data structure that supports a MultiPop operation in addition to the usual Push and Pop operations
  - The MultiPop operation takes an argument k that is required to be at most the number of items on the stack, and removes the top k items from the stack
  - The actual cost of a MultiPop operation is k, while the actual cost of a Push of Pop operation is 1
- We wish to upper bound the worst-case cost of any sequence of n operations starting from the empty stack
- The naive upper bound is $O(n^2)$ since c_i could be as high as $\max(1, i-1)$
- Intuition: For each "expensive" operation (cost k, say), we need to have about k "cheap" operations (cost 1)

Aggregate Method

- Suppose our sequence consists of x Push operations, y Pop operations, and $z = n - x - y$ MultiPop operations
- Let k denote the total number of items removed by the z MultiPop operations
- Thus the total cost is $x + y + k$
- Since $x \geq y + k$, the total cost is at most $2x$
- Since $x \leq n$, the total cost is at most $2n$
- Thus we can assign an amortized cost of 2 to each operation

Accounting Method

- We will maintain the invariant that there is one token associated with each item on the stack
- A PUSH costs one unit and leaves a token on the new item, so we can set its amortized cost to 2
- A POP costs one unit which can be paid for using the token associated with the item being removed, so we can set its amortized cost to 0
- A MULTIPOP with parameter k costs k units which can be paid for using the tokens associated with the k items being removed, so we can sets its amortized cost to 0
- Since each operation has an amortized cost of at most 2, the total cost of any sequence of n operations is at most $2n$

Potential Function Method

- We define $\Phi(D_i)$ as the number of items in the stack in state D_i
- If operation i is a PUSH, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$
- If operation i is a POP, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 - 1 = 0$
- If operation i is a MULTIPOP with parameter k, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k - k = 0$
- Since each operation has an amortized cost of at most 2, the total cost of any sequence of n operations is at most $2n$

Example: Incrementing a Binary Counter

- Assume that our "data structure" is a binary counter that is initially zero
- The only operation on our data structure is the increment operation
- Assume that the actual cost of an increment is equal to the number of bits flipped, e.g., the cost of incrementing 111010011 to 111010100 is 3
- In a sequence of n increments, the worst-case cost of a single increment operation is $\Theta(\log n)$, which implies an $O(n \log n)$ bound on the total cost of the sequence
- We'd like to use amortized analysis to obtain an $O(n)$ bound

Aggregate Method

- Every increment flips the low-order bit, so the number of times the low-order bit is flipped is n
- The second, fourth, sixth, ... increments flip the second-lowest order bit, so the number of times this bit is flipped is at most $\lfloor n/2 \rfloor \leq n/2$
- The fourth, eighth, twelfth, ... increments flip the third-lowest order bit, so the number of times this bit is flipped is at most $\lfloor n/4 \rfloor \leq n/4$
- Given this pattern, the total number of bit flips is at most

$$n\left(1 + \frac{1}{2} + \frac{1}{4} + \dots\right) \leq 2n$$

- Thus we can assign an amortized cost of 2 to each increment

Accounting Method

- We will maintain the invariant that there is a token associated with each 1 bit in the binary counter
  - Since the counter is initially zero, the initial number of tokens is zero
- Suppose the ith increment operation is in a state D_{i-1} where the number of trailing 1's is equal to k (i.e., the binary counter ends with 01^k)
  - The actual cost c_i is $k + 1$
  - The amortized cost is $k + 1 - k + 1 = 2$
- Since each operation has an amortized cost of 2, the total cost of any sequence of n operations is at most $2n$

Potential Function Method

- As in the case of the "stack with multipop" example, we can easily tranform our accounting method argument into a potential function method argument
- We define the potential $\Phi(D)$ as the number of 1 bits in binary counter D

Example: A Dynamic Table

- We wish to implement an one-dimensional table with an INSERT operation that appends a given element to the end of the table
- Initially, we allocate a 0-entry table
- The first time we insert an item, we increase the capacity of the table to 1 and then insert, at an actual cost of 1
- Subsequently, each time we insert an item, we check whether the table is full
  - If not, we insert the item into the appropriate entry at an actual cost of 1
  - If so, we allocate a new table of capacity $2k$ where k denotes the current capacity, copy over the k items, and insert the new item, at an actual cost of $k + 1$

Potential Function Method

- We wish to design a potential function $\Phi(D)$ so that the amortized cost of each insertion is constant
- We need to ensure that there is a large drop in potential whenever we have to allocate a new table
- Between such “expensive” operations, the potential should increase gradually
- Let $f(D)$ denote the number of elements in table D, and let $g(D)$ denote the capacity of table D
    - Note that $f(D_i) = f(D_{i-1}) + 1$
- It is natural to consider setting $\Phi(D)$ to a function of the form $a \cdot f(D) + b \cdot g(D)$ for suitable constants a and b
    - How should we choose a and b?

The Case of a “Cheap” Insertion

- Suppose insertion i does not involve allocating a new table, so $g(D_i) = g(D_{i-1})$
- Thus $\Phi(D_i) - \Phi(D_{i-1}) = a$
- By setting a to a positive constant (to be determined), we obtain the desired gradual increase in the potential as the table is filled
- The amortized cost of a cheap insertion is $a + 1$

The Case of an "Expensive" Insertion

- Suppose insertion $i > 1$ require us to allocate a new table, so $g(D_i) = 2g(D_{i-1})$
- Thus $\Phi(D_i) - \Phi(D_{i-1}) = a + b \cdot g(D_{i-1})$
- To get a drop in potential that offsets the $g(D_{i-1})$ cost of moving the elements in the old table to the new table, we can set b to -1
- The amortized cost of an expensive insertion is $a + 1$
- The amortized cost of the first insertion is $c_1 + \Phi(D_1) - \Phi(D_0) = 1 + (a - 1) - 0 = a$
- We need to ensure that the potential is nonnegative
  - The table is always at least half full
  - We can set a to 2, resulting in an amortized cost of at most 3 for each operation

Handling Deletions

- Suppose that we also wish to support deletion of the last entry in the table, while ensuring that the "load factor" of the table is constant, say between $1/4$ and $1/2$
- When a deletion would cause the table to become less than one-quarter full, we allocate a new table of half the current capacity, and copy the table entries over
  - If the table capacity before the deletion is 1, 2, or 4 (and so the number of entries in the table is 1), the new table capacity is 0 and the actual cost is defined to be 1
  - Otherwise, the table capacity before the deletion is 2^k for some positive integer $k > 2$ (and so the number of entries in the table is 2^{k-2}), the new table capacity is 2^{k-1} and the actual cost is defined to be $1 + (2^{k-1} - 1) = 2^{k-1}$
- When a deletion does not cause the table to become less than one-quarter full, we simply delete the last item in the table at an actual cost of 1

Designing the Potential Function

- When $f(D) \geq g(D)/2$, we'd like the potential function to behave as in the insertion-only case
  - Increase gradually from zero to $g(D)$ as the table fills up
  - Drop close to zero when the table capacity is doubled
- When $g(D)/4 \leq f(D) < g(D)/2$, we'd like the potential function to behave in an analogous manner
  - Increase gradually from zero to $g(D)/4$ as the table shrinks
  - Drop close to zero when the table capacity is halved

The Potential Function

- When $f(D) \geq g(D)/2$, we define the potential function exactly as in the insertion-only case
- Otherwise, we define $\Phi(D)$ as $g(D)/2 - f(D)$
- It is not difficult to verify that the amortized cost of each operation is at most constant
- We conclude that the total cost of any sequence of n operations is $O(n)$

Remarks on Potential Function Arguments

- As we have seen, potential function arguments can be useful for bounding the total cost of a sequence of operations
- Potential function arguments are sometimes used for other purposes, such as for analyzing the quality of the output of an algorithm
- Let's take a look at an example of the latter type

Prediction from Expert Advice

- Each day, we need to make one yes/no decision
  - After we make our decision, we immediately find out whether or not it is correct
  - Before we make our decision, each of n experts tells us whether they think the correct decision is “yes” or “no”
  - We make no assumptions concerning the accuracy of the experts
- We'd like to ensure that the number of mistakes we make isn't too much worse than the number of mistakes made by the “best” expert (i.e., the one making the fewest mistakes)
- To what extent is such “regret minimization” possible?
  - We will restrict attention to deterministic strategies

The Majority Algorithm

- One natural idea is to make our decision according to the majority of the expert suggestions
- Suppose a third of the experts are always right, and two-thirds are always wrong
- If we follow the majority, we will always be wrong, even though some of the experts are never wrong
- So this simple strategy fails
- Instead, we will maintain a positive weight w_i for each expert i, $1 \leq i \leq n$
  - On a given day, the experts who have made fewer mistakes up to that day will have a higher weight
  - We will make our decision according to the weighted majority

The Weighted Majority Algorithm

- Let ε be a positive constant less than $1/2$
- For any expert i and day t, we determine the weight $w_i^{(t)}$ assigned to the prediction of expert i on day t as follows
  - We set $w_i^{(1)}$ to 1
  - If expert i is correct on day t, we set $w_i^{(t+1)}$ to $w_i^{(t)}$
  - If expert i is incorrect on day t, we set $w_i^{(t+1)}$ to $(1-\varepsilon)w_i^{(t)}$
  - Thus $w_i^t = (1-\varepsilon)^k$ where k denotes the number of mistakes made by expert i prior to day t
- On day t, we use the weighted majority of the expert predictions to determine whether to say "yes" or "no"

Analysis of the Weighted Majority Algorithm

- Let us define the potential function $\Phi^{(t)}$ as $\sum_{1 \le i \le n} w_i^{(t)}$
- Lemma: If we make a mistake on day t, then $\Phi^{(t+1)} \le (1 - \varepsilon/2)\Phi^{(t)}$
  - In a weighted sense, at least half of the experts are wrong on day t
  - Thus, at least half of the expert weight is knocked down by a factor of $1 - \varepsilon$
  - Thus $\Phi^{(t)} - \Phi^{(t+1)} \ge (\varepsilon/2)\Phi^{(t)}$
- Let r denote the number of mistakes that we make in the first T days, and let s denote the minimum number of mistakes made by any of the experts in the same timeframe

Analysis of the Weighted Majority Algorithm (cont'd)

- Theorem: We have $r \leq 2s(1+\varepsilon) + (2/\varepsilon)\ln n$
- The lemma implies $\Phi^{(T+1)} \leq n(1-\varepsilon/2)^r$
- The definition of s implies that $\Phi^{(T+1)} \geq (1-\varepsilon)^s$
- Thus $(1-\varepsilon)^s \leq n(1-\varepsilon/2)^r$, and taking logarithms we obtain $s\ln(1-\varepsilon) \leq r\ln(1-\varepsilon/2) + \ln n$
- We have $\ln(1+x) = x - x^2/2 + x^3/3 - \ldots$ for $|x| < 1$
  - Thus $\ln(1-\varepsilon/2) \leq -\varepsilon/2$
  - And it is easy to argue that $\ln(1-\varepsilon) \geq -\varepsilon - \varepsilon^2$ since $\varepsilon < 1/2$
- Thus $s(\varepsilon + \varepsilon^2) \geq r(\varepsilon/2) - \ln n$, and hence $s(1+\varepsilon) + \ln n \geq r/2$; the theorem follows